ACTIVE DEBUGGING ENVIRONMENT FOR APPLICATIONS CONTAINING COMPILED AND INTERPRETED PROGRAMMING LANGUAGE CODE

RELATED APPLICATIONS

This Application is a Continuation-in-Part of Application serial number 08/815,719 filed March 12, 1997, the text of which is incorporated by reference to the same extent as if the text where present herein.

COPYRIGHT AUTHORIZATION

FIELD OF THE INVENTION

This invention relates to programming language debugging tools, and in particular, to an active debugging environment that is programming language neutral and host neutral for use in debugging any of a variety of disparate compiled and/or interpreted programming languages that may exist individually or in combination within a given application.

PROBLEM

The end user of a modern software product sees and uses an application that was created by an application developer. For purposes of this discussion, an application is a computer program written in any of a plurality of programming languages. Typically, a computer program contains some original program code and certain pre-existing or "canned" components that include, but are not limited to, modules, libraries, sub-routines, and function calls. The pre-existing application components are used where ever possible to limit the number of mistakes that are introduced into the application during development, and to minimize the amount of overall effort required to create the application.

As the number of applications and their components have increased, and the number of different programming languages used to generate the

applications have increased, so also has the need grown for applications to expose their internal services to other applications in a consistent manner independent of the underlying programming languages involved. This need for exposing internal services of various applications in a universal manner is referred to as providing programmability.

Existing program architectural models such as the Component Object Model (COM) have greatly contributed to programmability across different applications. The COM model establishes a common paradigm for interactions and requests among different applications regardless of the programming languages or components involved. With COM, for example, an Internet web page application can be created that calls on only certain of the components of a word processing application, a spread sheet application, and a database application, that are needed to complete the web page application without including the entire bulk of each of the word processing, spread sheet, and database features within the web page application.

However, one problem that results from creating an application that includes multiple program components from many different programming language sources, is debugging. A first and third program component in an application may have originated from two different compiled language sources and a second and fourth program component in an application may have originated from two different interpretive language sources. Not only is there historically a fundamental difference in the implementation of a debugger for a compiled programming language versus an interpreted programming language, each programming language can have its own proprietary interfaces and other features that make debugging the aggregate application a difficult and/or impossible task.

For purposes of this document, a compiled programming language is considered a native machine code compilable programming language having a specific target platform. Examples of compiled programming languages include, but are not limited to, C and C++. Alternatively, an interpreted programming language is a run-time bytecode interpreted or source code interpreted programming language that operates under control of a master within a given application. Examples of interpreted programming languages include Visual Basic, Visual Basic for Applications (VBA), Visual Basic Script, Java, JavaScript, Perl, and Python. The Java programming language is included in the category

of interpreted programming languages for purposes of this document even though Java is compiled from source code to produce an object and there is no access to the Java source during run time. One key reason Java is included is because the compiled Java object is fundamentally a bytecode object that requires a language engine rather than the traditional machine code link, load, and execute steps.

One example of a compiled programming language debugger limitation is that they require knowledge of the static environment from which the run-time object code was generated. The static environment of a compiled programming language includes the source code and the corresponding object code. A debugger for a compiled programming language performs the work of generating a mapping of the structures between the source code and the object code prior to executing the object being debugged, and the debugger requires that the source code and object code remain consistent throughout the debug process. Any changes to either the source code and/or the object code render the debug mapping and subsequent debugging capability unsound. For this reason, compiled programming language debuggers do not tolerate run time changes to code and the debugger for one compiled programming language is not functional for any other programming language.

Alternatively, interpreted programming languages are more flexible in that they are run-time interpreted by a programming language engine that does not require a static source code or object code environment. However, interpreted programming language debuggers do not accommodate compiled programming language debugging and are often functional with only one interpreted programming language.

Another problem with compiled programming language debuggers is that they only function under known predefined run-time conditions with a specific operating environment. However, even under these constraints existing compiled programming language debuggers are only aware of predefined host application content that is made available to the debugger prior to run time, but the debuggers have no run-time knowledge of host application content.

One solution to the difficulty with debugging applications that contain disparate code from compiled programming languages and/or interpreted programming languages is to limit the developer to using only one programming language for an application. However, this solution is undesirable because no

one programming language is ideal for every application. Further, this solution is unreasonable because the demands of present day Internet web page programming, as well as the general customer/developer demands in the computing industry, require multi-language extensibility.

For these reasons, there exists an ongoing need for a debugging technology that facilitates efficient programmability by way of programming language, host application, and operating environment independence. A system of this type has heretofore not been known prior to the invention as disclosed below.

SOLUTION

The above identified problems are solved and an advancement achieved in the field of programming language debuggers due to the active debugging environment of the present invention for applications containing compiled and interpreted programming language code. The active debugging environment facilitates content rich run-time debugging in an active debug environment even if a mixture of compiled and interpreted programming languages exist within the application being debugged. One purpose of the active debugging environment is to provide an open and efficiently deployed framework for authoring and debugging language neutral and host neutral applications.

In the context of the present discussion, language neutral means that a debugging environment exists that transparently supports multi-language program debugging and cross-language stepping and breakpoints without requiring specific knowledge of any one programming language within the environment. Host neutral, also referred to as content-centric, means that the debugging environment can be automatically used with any active scripting host such that the host application has control over the structure of the document tree presented to the debug user, including the contents and syntax of coloring of the documents being debugged. In addition, the debugging environment provides debugging services that include, but are not limited to, concurrent host object model browsing beyond the immediate run-time scope. Host document control allows the host to present the source code being debugged in the context of the host document from which it originated. Further, the language neutral and content-centric host neutrality, facilitates a developer transparent debugging environment having multi-language extensibility, smart host document and

application context management, and virtual application discoverability and dynamicness.

Discoverability means the ability for the debugging environment to be started during program run-time and immediately step into a running application with full knowledge of the executing program's context and program execution location. Dynamicness means the concept of flexible debugging where there is no static relationship between the run-time environment at the beginning of program execution and the run-time environment at some later point in program execution. In other words, script text can dynamically be added to or removed from a running script with the debugging environment having immediate and full knowledge of the changes in real time.

Key components of the active debugging environment include, but are not limited to, a Process Debug Manager (PDM) that maintains a catalog of components within a given application, and a Machine Debug Manager (MDM) that maintains a catalog of applications within a virtual application. The active debugging environment components work cooperatively with any replaceable and/or generic debug user interface that support typical debugging environment features. The active debugging environment also cooperatively interacts with a typical active scripting application components that include, but are not limited to, at least one language engine for each programming language present in a given script, and a scripting host.

The method for debugging a multiple language application in an active debugging environment includes, but is not limited to, defining a content centric host, defining a language neutral debugging environment, generating a virtual application that includes the multiple compiled and interpretive programming language statements and related programming language context, and executing the virtual application on the content centric host under control of the language neutral active debugging environment.

Defining a content centric host includes establishing a language engine component for each unique programming language associated with the multiple compiled and/or interpreted programming language statements. In addition, the language engine component includes programming language specific mapping and debugging features. Defining the content centric host further includes coordinating in-process activities of the content centric host with each of the language engine component and the language neutral debugging environment.

Defining an active debugging environment includes establishing a PDM and an MDM as the core components to facilitate debugging by way of a variety of existing language engines and debug user interfaces, and to coordinate language neutral and host neutral communications between the active debugging environment and the content centric host during debug operations.

Additional details of the present invention will become apparent and are disclosed in the text accompanying FIGs. 1-7 as set forth below. Appendix A is included to disclose specific details of active scripting interfaces used in one example of a run-time implementation. Appendix B is included to disclose specific details of interfaces used in one example of an active debugging environment implementation.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates an example of a computing system environment example in block diagram form on which the claimed invention can be implemented;

FIG. 2 illustrates a standard object interface example in block diagram form;

FIG. 3 illustrates a scripting architecture overview and operational example in block diagram form;

FIG. 4 illustrates an active debugging environment example in block diagram form;

FIG. 5 illustrates an overview of the active debugging operational steps in flow diagram form;

FIG. 6 illustrates operational details of the run time environment for a virtual application in flow diagram form; and

FIG. 7 illustrates details of the active debugging environment operational steps in flow diagram form.

DETAILED DESCRIPTION

Computing System Environment - FIG. 1

FIG. 1 illustrates an example of a computing system environment 100 on which the claimed invention could be implemented. The computing system environment 100 is only one example of a suitable computing environment for the claimed invention and is not intended to suggest any limitation as to the scope of use or functionality of the claimed invention. Neither should the

computing environment 100 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary computing system environment 100.

The claimed invention is operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the claimed invention can include, but are also not limited to, a general purpose Personal Computer (PC), hand-held or lap top computers, multi-processor systems, microprocessor-based systems, programmable consumer electronics, network computers, Personal Communication Systems (PCS), Personal Digital Assistants (PDA), minicomputers, mainframe computers, distributed computing environments that include any one or more of the above computing systems or devices, and the like.

The claimed invention may also be described in the general context of computer-executable instructions that are executable on a PC. Such executable instructions include the instructions within program modules that are executed on a PC for example. Generally, program modules include, but are not limited to, routines, programs, objects, components, data structures, and the like that perform discrete tasks or implement abstract data types. The claimed invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory devices.

The exemplary computing system environment 100 is a general purpose computing device such a PC 110. Components of PC 110 include, but are not limited to, a processing unit 120, a system memory 130, and a system bus 121. The system bus 121 communicatively connects the aforementioned components and numerous other cooperatively interactive components.

Processing unit 120 is the primary intelligence and controller for PC 110 and can be any one of many commercially available processors available in the industry. System bus 121 may be any combination of several types of bus structures including, but not limited to, a memory bus, a memory controller bus, a peripheral bus, and/or a local bus. System bus 121, also referred to as an expansion bus or I/O channel, can be based on any one of a variety of bus

architectures including, but not limited to, Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA), Enhanced ISA (EISA), Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) also known as Mezzanine bus.

System memory 130 is a volatile memory that can include a Read Only Memory (ROM) 131 and/or a Random Access Memory (RAM) 132. ROM 131 typically includes a Basic Input/Output System (BIOS) 133. BIOS 133 is comprised of basic routines that control the transfer of data and programs between peripheral non-volatile memories that are accessible to PC 110 during start-up or boot operations. RAM 132 typically contains data and/or programs that are immediately accessible to and/or presently being operated on by processing unit 120. Types of data and/or programs in RAM 132 can include operating system programs 134, application programs 135, other program modules 136, and program data 137.

Other components in PC 110 include numerous peripheral devices that are accessible to processing unit 120 by way of system bus 121. The numerous peripheral devices are supported by appropriate interfaces that can include a first non-volatile memory interface 140 for non-removable non-volatile memory device support, a second non-volatile memory interface 150 for removable non-volatile memory device support, a user input interface 160 for serial device support, a network interface 170 for remote device communication device support, a video interface 190 for video input/output device support, and an output peripheral interface 195 for output device support.

Examples of a non-removable non-volatile memory device can include a magnetic disk device 141 or other large capacity read/write medium such as an optical disk, magnetic tape, optical tape, or solid state memory. Types of data often stored on a non-removable non-volatile memory device include persistent copies of programs and/or data being used and/or manipulated in RAM 132 such as operating system programs 144, application programs 145, other program modules 146, and program data 147.

One example of a removable non-volatile memory device can include a magnetic floppy disk device or hard disk device 151 that accepts removable magnetic media 152. Another example of a removable non-volatile memory device can include an optical disk device 155 that accepts removable optical media 156. Other types of removable media can include, but are not limited to,

magnetic tape cassettes, flash memory cards, digital video disks, digital video tape, Bernoulli cartridge, solid state RAM, solid state ROM, and the like.

User input interface 160 supports user input devices that can include, but are not limited to, a pointing device 161 commonly referred to as a mouse or touch pad, and a keyboard 162. Other user input devices can include, but are not limited to, a microphone, joystick, game pad, neuro-stimulated sensor, and scanner, and may require other interface and bus structures such as a parallel port, game port or a Universal Serial Bus (USB) for example.

User input/output devices supported by video interface 190 can include a display monitor 191 or a video camera. Output peripheral interface 195 supports output devices such as printer 196 and speakers 197.

Network interface 170 supports communications access to a remote computing facility such as remote computer 180 by way of Local Area Network (LAN) 171 and/or Wide Area Network (WAN) 173, or other Intranet or Internet connection. Other remote computing facility types for remote computer 180 can include, but are not limited to, a PC, server, router, printer, network PC, a peer device, or other common network node. A remote computer 180 can typically include many or all of the components described above for PC 110. Modulator/Demodulator (MODEM) 172 can also be used to facilitate communications to remote computer 180. Types of programs and/or data accessible from remote memory device 181 on remote computer 180 can include, but are not limited to, remote application programs 185.

Definitions

The following terms and concepts are relevant to this disclosure and are defined below for clarity.

Term	Definition
Code Context	A representation of a particular location in running code of a language engine, such as a virtual instruction pointer.
Code object	An instance created by a script host that is associated with a named item, such as the module behind a form in Visual Basic, or a C++ class associated with a named item.
Context	An abstraction also referred to as document context that

	represents a specific range in the source code of a host document.
Debug Event	A run-time flow altering condition, such as a breakpoint, that is set in a virtual application by a debugging environment user and managed by the active debugging environment that controls the running of the virtual application. Debug events are defined by the active debugger.
Debugger IDE	A replaceable component of the active debugging environment that supports an Integrated Development Environment (IDE) debugging User Interface (UI), and the primary component that communicates with the host application and one or more of the language engines.
Expression Context	A context of a resource, such as a stack frame, in which expressions may be evaluated by a language engine.
Host Application	The application or process that hosts one or more language engines and provides a scriptable set of objects, also known as an object model.
Language Engine	A replaceable component of the active debugging environment, sometimes referred to as a scripting engine, that provides parsing, execution, and debugging abstractions for a particular language, and implements IactiveScript and/or IactiveScriptParse.
Machine Debug Manager	A key component of the active debugging environment that maintains a registry of debuggable application processes.
Named item	An object, such as one that supports OLE Automation, that the host application deems interesting to a script. Examples include the HTML Document object in a Web browser or the Selection object in Microsoft Word.
Object Browsing	A structured, language-independent representation of an object name, type, value, and/or sub-object that is suitable for implementing a watch window user interface.
Process Debug Manager	A key component of the active debugging environment that maintains a registry of components for a given application that

include the tree of debuggable documents, language engines, running threads, and other application resources.

Script — A set of instructions that are run by a language engine. A script can include, but is not limited to any executable code, piece of text, a block of pcode, and/or machine-specific executable byte codes. A script is loaded into a language engine by a script host by way of one of the Ipersist* interfaces or the IactiveScriptParse interface.

Script language — An interpreted programming language and the language semantics used to write a script.

Script host — An application or program that owns the ActiveScripting engine. The scripting host implements IactiveScriptSite and optionally IactiveScriptSiteWindow.

Scriptlet — A portion of a script that is attached to an object event through IactiveScriptParse. An aggregate of scriptlets is a script.

Standard Object Interface Example - FIG. 2

FIG. 2 illustrates a standard object interface example 200 in block diagram form. The standard object interface example 200 includes a local computing device 201 and a remote computing device 202, both of which can be personal computers. The remote computing device 202 includes a remote operating system 240 and a host process 250. The host process 250 includes at least one application object 251 having multiple interfaces including but not limited to interfaces 252-254. Each interface 252-254 is a standard COM interface that each exposes at least one method.

The local computing device 201 includes a local operating system 210, a first host process 220 and a second host process 230. The local operating system 210 includes a plurality of application objects 211-212 each having at least one of a plurality of interfaces 214-218. The second host process 230 includes a plurality of application objects including 232-233 each having at least one the plurality of interfaces 234-238. The first host process 220 includes, but is not limited to, a plurality of local and/or remote interface calls 221-226 to respective interfaces in host processes 250, 230, and local operating system 210 as indicated by the directional arrows away from the first host process 220. The

interface calls 221-226 occur at various points in a script 228 and two of the calls 222 and 226 result in the downloading of a small section of a foreign program into the script 228 for execution. The downloaded programs may have originated in any compiled or interpreted programming language other than the programming language of script 228. Key to each of the above referenced interface calls is that they are calls to common interfaces that are made in a common manner from any application or process.

Scripting Architecture Overview and Operational Example – FIG. 3

A scripting language engine interface provides the capability to add scripting and OLE Automation capabilities to programs such as applications or servers. One example of a scripting language engine interface is embodied in the commercially available product known as ActiveX™ Scripting by Microsoft. ActiveX Scripting enables host computers to call upon disparate language engines from multiple sources and vendors to perform scripting between software components. The implementation of a script itself including the language, syntax, persistent format, execution model, and the like, is left to the script vendor. Care has been taken to allow host computers that rely on ActiveX Scripting to use arbitrary language back-ends where necessary.

Scripting components fall into at least two categories that include script hosts and language engines. A script host creates an instance of a language engine and calls on the language engine to run at least one portion of a script. Examples of script hosts can include but are not limited to Internet and Intranet browsers, Internet and Intranet authoring tools, servers, office applications, and computer games. A desirable scripting design isolates and accesses only the interface elements required by an authoring environment so that non-authoring host computers such as browsers and viewers, and the associated language engines can be kept relatively compact.

FIG. 3 illustrates an operational active scripting architecture example 300 in flow diagram form. Examples of active scripting interfaces are disclosed in Appendix A. Components of the operational active scripting architecture example 300 include a language engine 301, a host document or application 305, a script 304, and various object interfaces 310-314 for the language engine 301 and host application 305 respectively. The following discussion illustrates the interactions between the language engine 301 and the host application 305.

First, a host application 305 begins operations by creating an instance of

the application in a workspace of a computing device. A copy of the host application 305 can be obtained from a storage device 306 or other source in any manner well known in the art.

Second, the host application 305 creates an instance of the language engine 301 by calling the function CoCreateInstance and specifying the class identifier (CLSID) of the desired language engine 301. For example, the Hyper Text Markup Language (HTML) browsing component of Internet Explorer receives the language engine's class identifier through the CLSID= attribute of the HTML <OBJECT> tag. The host application 305 can create multiple instances of language engine 301 for use by various applications as needed. The process of initiating a new language engine is well known in the art.

Third, after the language engine 301 is created, the host application 305 loads the script 304 itself into the language engine 301 by way of interface/method 310. If the script 304 is persistent, the script 304 is loaded by calling the IPersist*::Load method 310 of the language engine 301 to feed it the script storage, stream, or property bag that is resident on the host application 305. Loading the script 304 exposes the host application's object model to the language engine 301. Alternatively, if the script 304 is not persistent then the host application 305 uses IPersist*::InitNew or IActiveScriptParse::InitNew to create a null script. As a further alternative, a host application 305 that maintains a script 304 as text can use IActiveScriptParse::ParseScriptText to feed the language engine 301 the text of script 304 after independently calling the function InitNew.

Fourth, for each top-level named item 303 such as pages and/or forms of a document that are imported into the name space 302 of the language engine 301, the host application 305 calls IActiveScript::AddNamedItem interface/method 311 to create an entry in the name space 302. This step is not necessary if top-level named items 303 are already part of the persistent state of the previously loaded script 304. A host application 305 does not use AddNamedItem to add sublevel named items such as controls on an HTML page. Instead, the language engine 301 indirectly obtains sublevel items from top-level items by using the ITypeInfo and/or IDispatch interface/method 314 of the host application 305.

Fifth, the host application 305 causes the language engine 301 to start running the script 304 by passing the SCRIPTSTATE_CONNECTED value to

the IActiveScript::SetScriptState interface/method 311. This call typically executes any language engine construction work, including static bindings, hooking up to events, and code execution similar to a scripted "main()" function.

Sixth, each time the language engine 301 needs to associate a symbol with a top-level named item 303, the language engine 301 calls the IActiveScriptSite::GetItemInfo interface/method 312. The, the IActiveScriptSite::GetItemInfo interface/method 312 can also return information about the named item in question.

Seventh, prior to starting the script 304 itself, the language engine 301 connects to the events of all relevant objects through the IConnectionPoint interface/method 313. For example, the IConnectionPoint::Advise(pHandler) message provides the language engine 301 with a request for notification of any events that occur in the host application 305. The IConnectionPoint::Advise message passes an object pHandler that can be called when an event occurs in the host application 305. Once an event occurs in the host application 305, the host application 305 transmits a message to the language engine 301 pdispHandler::Invoke(dispid) as notice that an event occurred in the host application 305. If the event that occurred matches an event that is being monitored by the language engine 301, the language engine 301 can activate a predetermined response.

Finally, as the script 304 runs, the language engine 301 realizes references to methods and properties on named objects through the IDispatch::Invoke interface/method 314 or other standard COM binding mechanisms. Additional implementation specific details of ActiveX Scripting interfaces and methods are disclosed Appendix A.

One purpose of ActiveX is to allow a developer to expose internal objects and/or properties of an application as interfaces and methods available to other applications. A method is an action which the object can perform, and a property is an attribute of the object similar to a variable. The ActiveX interfaces include IDispatch which is an interface to manipulate ActiveX objects. This process is used to get a property, set a property, or call a method. The process uses a late binding mechanism that enables a simple non-compiled interpretive language. Type information in ActiveX includes ITypeInfo which is used for describing an object. A collection of these TypeInfos constitutes a type library, which usually exists on a disk in the form of a data file. The data file can be accessed through

ITypeLib and is typically created using MKTypLib. In ActiveX scripting, the type information is provided by scripting hosts and objects that are used by the scripting hosts.

Active Debugging Environment – FIG. 4

FIG. 4 illustrates an example of an active debugging environment 400 in block diagram form based on the standard object interface example 200 of FIG. 2. In the active debugging environment 400 example, the first host process 220 contains the application 421 that is the debugging target although any application in one of the host processes 220, 230, or 250 can be the debugging target if desired. For example, the host process 250 might include an Internet web page application on an Internet server 202, and the host process 220 might include an Internet browser application under development on an end user's local machine 201. The Internet browser application under development would be the debugging target so that the application developer can watch what is happening as the browser interacts with the remote web page and exercises various features and controls of the web page.

Components of the overall active debugging environment 400 include active scripting application components 420, key debugging environment interface components that include the PDM 424 and the MDM 411, and the IDE 410 debug interface. It is important to note that the active scripting application components 420 are standard application components of a product, such as an Internet browser, that are shipped to and used by an end user upon general release of a version of the product. That the standard application components are designed with active debugging environment interfaces in mind is transparent to the end user of the application. Note also that the IDE 410 is an active debugging component that is replaceable by any developer wishing to architect a debugging interface. The PDM 424 and MDM 411 are the fundamental components of the active debugging environment 400 with which the active scripting application components 420 and the IDE 410 must interact to facilitate a functional debugging environment.

The IDE 410 is a debug user interface between the debug user and the active debugging environment 400. The typical IDE 410 allows a debug user to engage in real time document and/or application editing, requesting various views of the running application, defining breakpoints and other execution control

management features, requesting expression evaluation and watch windows, and browsing stack frames, objects, classes, and application source code.

One example of debug interfaces for an IDE 410 are disclosed in Appendix B. The architecture of an IDE 410 can be designed to support features such as CModule, CBreakpoint, CApplicationDebugger, and CurStmt. For example, there can be one CModule instance per document/application being debugged such that the instance is maintained in a doubly-linked list headed by g_pModuleHead. Entries in the list can be displayed to the debug user in a selectable Module menu and any selected list item would result in document text being displayed in the richedit control. From the richedit control, the debug user could define set breakpoints or user other viewing features. If a debug user were to set a breakpoint, one Cbreakpoint instance would be generated and maintained in a doubly-linked list headed by the g_pBpHead field of the associated CModule. Each breakpoint position could be represented to the debug user as a text range of the document. One CApplicationDebugger instance would be generated and maintained by the application that implements the IApplicationDebugger interface so that the CApplicationDebugger could respond to all debug events that occur. Finally, the CurStmt could hold the location of the current statement being executed in addition to referencing the thread associated with the present breakpoint.

The Machine Debug Manager (MDM) 411 maintains a list of active virtual applications and is a central interface between the IDE 410 and the active script components 420. Virtual applications are collections of related documents and code in a single debuggable entity such that separate application components in a continuous line of code can share a common process and/or thread. A virtual application is the aggregate of multiple applications in multiple programming languages. One key role of the machine debug manager is to act as a program language registry that provides a mapping between a given application in the virtual application aggregate and the active debugger IDE 410 that is controlling the virtual application during the debug process. The MDM 411 eliminates the traditional debugging model where the debugger for a given programming language only has knowledge of a specific source and object code mapping. Instead, the MDM 411 places the burden of language specific details on the programming language to determine, for example, the mapping of a breakpoint requested by the debug user to a specific instruction in the programming

language code within a given application. For an interpreted programming language, a programming language specific decision is passed through the PDM 424 to an appropriate language engine 422-423. For a compiled programming language, a programming language specific decision is passed through the PDM 424 to a library of source and object codes associated with an appropriate language engine 422-423.

The active scripting application components 420 include a script host 421 and at least one language engine 422-423, and each of the at least one language engine 422-423 operate in close cooperation with the PDM 424. The PDM 424 includes a catalog of components within a given application, and acts as an interface for all in-process activities and synchronizes the debugging activities of the multiple language engines 422-423 such as merging stack frames and coordinating breakpoints for example. The PDM 424 also maintains a debugger thread for asynchronous processing and acts as the communication interface for the MDM 411 and the IDE 410.

A language engine 422-423 supports a specific language implementation and provides specific language features used by the corresponding program language code in the virtual application. Among the specific language features are the breakpoint, start, stop, and jump debug implementations, expression evaluations, syntax coloring, object browsing, and stack frame enumeration. Each language specific feature is unique to a programming language regardless of the quantity of that programming language that exists in the virtual application.

The application or script host 421 provides the environment for the language engines 422-423. Additional tasks of the script host 421 include supporting an object model, providing context for scripts, organizing scripts into virtual applications, maintaining a tree of documents being debugged and their contents, and providing overall document management. An example of a script host is an Internet browser such as Internet Explorer or Spruuids. One important script host concept is that a host does not necessarily require a design that has debugging in mind because it is the programming language that must have been created with debugging in mind. For this reason, the script host 421 does not require specific knowledge of any language specific debugging features. Specific knowledge of language specific debugging features is the task of the individual language engines 422-423 and/or their associated libraries.

It is also important to note that depending on the design of a script host 421, the script host 421 can exist in the active debugging environment 400 as a smart host or a dumb host. A dumb host is not aware of all available debug interfaces because all scripts are separate and/or are managed by language engines independently of each other, and no contextual view of source code exists. Nevertheless, a dumb host can support self-contained languages such as Java and can provide a default virtual application for the current process. Alternatively, a smart host is aware of the debug interfaces and can group program code and documents into a single virtual application to provide visual context to embedded languages such as HTML. The smart host takes advantage of and supports integrated debugging support across a mixed model of programming languages and is generally aware of the context and origin of individual scripts. Although, a smart host provides a more robust and efficient debugging environment, debugging can proceed in kind for a dumb host even if with somewhat limited flexibility.

Active Debugging Environment Operational Steps – FIGs. 5-7

FIG. 5 illustrates an example overview of active debugging environment operational steps 500 in flow diagram form. The active debugging environment operational steps 500 begin at step 508 and represent a high-level view of steps involved in the setup and operation of an active debugging environment where programming language code from multiple compiled and/or interpreted programming languages are present in the same virtual application debug target.

At step 512, the script host 421 generates a virtual application for run time execution. The virtual application can contain program language code from only single programming language, or the virtual application can contain an aggregate of programming language code from multiple compiled and/or interpreted programming languages. Additional details of step 512 are disclosed in the text accompanying FIG. 6.

At step 521, an active debugging environment is established that controls the stepwise flow of a run time script. Key to the active debugging environment 400 being programming language neutral and content-centric host neutral is that the programming language debug specifics are embedded in the active scripting application components 420 rather than in the debugging tool itself. Key responsibilities of the active debugging environment 400 is to facilitate programming language neutral and host neutral debugging by interfacing the

IDE 410 user interface with the active scripting application components 420 by way of the catalog of applications maintained in the MDM 411 and the catalog of application components in the PDM 424. The MDM 411 and the PDM 424 determine which application and which language engine 422-423 is responsible for a given section of program language code in the virtual application as previously disclosed in the text accompanying FIG. 4.

At step 530, with the run time environment established and the active debug environment in place, the previously generated virtual application is executed under the control of the active debugging environment 400. Note that during run time of the virtual application at step 538, that the active debugging environment 400 accommodates the dynamic run-time modification of program code in the script without disrupting script execution flow or the active debugging operations. One reason this dynamic run-time environment is possible even with the existence of compiled programming language segments present, is that each programming language segment added to or removed from the running script is accompanied by the necessary language engine support to assist with the language specific debug details.

At step 550, a debug user or application developer interacts with the active debugging environment during the debugging process by way of the IDE 410. Additional details of the debugging process interactions are disclosed in the text accompanying FIG. 7. The active debug operational steps 500 end at step 560 at the time the application developer chooses to stop the debugging process.

FIG. 6 illustrates details of the active scripting run-time environment operational steps 600 for a virtual application in flow diagram form. The active scripting run-time environment operational steps 600 begin at step 608 and represent the details of step 512 of FIG. 5. Although the text accompanying FIG. 6 discloses the details of one type of run-time environment for virtual applications, the active debugging environment 400 of the present invention is host neutral and functions equally well with any COM designed active scripting run-time environment.

At step 612, an instance of a script host 421 is created in an application 220 to manage run-time execution of a virtual application. Along with the script host 421, at least one programming language engine 422-423 is generated for each programming language used in an application. At step 622, the multiple

objects within the application are combined into a single virtual application, also known as script text. At step 630 the single virtual application is loaded into the script host 421 for run-time execution. Processing continues at step 638 by returning to step 512 of FIG. 5.

FIG. 7 illustrates details of debugging operational steps 700 in flow diagram form in the context of the active debugging environment 400. The debugging operational steps 700 begin at step 708 and represent the details of step 550 in FIG. 5. At step 712, an active debugging environment 400 is established that includes an IDE 410 as a user interface, a MDM 411 to catalog applications that are present and manage language specific interactions with each target application, and a PDM 424 to catalog application components and manage component specific activities during the debug process. At step 720, the virtual application script and the script host 421 are identified and the virtual application is run under the control of the active debugging environment 400. Starting a virtual application under the control of an active debugging environment means that breakpoints, stepwise code execution, and/or other script flow altering activities can be set at will by a human user to alter the run-time script flow as desired during the debug process.

At step 728, one or more event monitoring cases can be set in the active debugging environment 400 by communicating the event criteria to the appropriate active scripting application components 420 by way of the MDM 411 and/or PDM 424, and the IDE 410 user interface. Each event is specific to a given portion of the run-time script text even though the script text programming language details are transparent to the debug user. Further, the number of event monitoring cases defined for a given virtual application can be dynamically altered as different programming language code sections are downloaded to the script host 421 to dynamically become part of the running virtual application.

If it is determined at decision step 735 that the run-time script text has not reached an event being monitored, then the run-time script text continues running at step 735. In the mean time, the debug user is free to carry out other activities in the context of the active debugging environment that can include, but are not limited to, viewing other documents, the document tree, source code, or even make changes to the source code as desired. Alternatively, if it is determined at decision step 735 that the run-time script text has reached an event being monitored, then processing continues at step 742. At step 742, a

predefined debugging response is activated in response to the occurrence of the monitored event. Predefined responses can include, but are not limited to, displaying a range of script text that contains a breakpoint, displaying variable or other reference contents, and displaying stack pointers, all within the context supported by the corresponding language engine specific to the immediate programming language statements. All communications between the debug user and the script host 421 are facilitated by the MDM 411 and/or the PDM 424 that process and pass requests to an appropriate language engine 422-423 for a response.

If it is determined at decision step 755 by the debug user that the debugging process should continue, then processing continues at step 728 as previously disclosed. Alternative, if it is determined at decision step 755 by the debug user that the debugging process is complete, then the debug communicates to the IDE 410 that debug processing should stop and processing continues at step 762 by returning to step 550 of FIG. 5.

Conclusion

The present invention is an active debugging environment that is programming language independent and host application independent for use in debugging a mixture of compiled and interpreted programming languages in a given application. The programming language and host application independence is facilitated by Machine Debug Manager and Process Debug Manager interfaces that identify an appropriate language engine and/or application specific components that is responsible for unique programming language details. Appendices are attached to this document that disclose examples of active run-time script engine interfaces and active debugging environment interfaces. Specific active run-time script implementation interfaces are disclosed in Appendix A. Specific active debugging environment implementation interfaces are disclosed in Appendix B.

Although specific embodiments are disclosed herein, it is expected that persons skilled in the art can and will make, use, and/or sell alternative active debugging environment systems that are within the scope of the following claims either literally or under the Doctrine of Equivalents.